Implementation and Performance of Probabilistic Inference Pipelines: Data and Results

Dimitar Shterionov
Gerda Janssens

Report CW 679, March 2015

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Implementation and Performance of Probabilistic Inference Pipelines: Data and Results

Dimitar Shterionov
Gerda Janssens

Report CW 679, March 2015

Department of Computer Science, KU Leuven

Abstract

In order to handle real-world problems, state-of-the-art probabilistic logic and learning frameworks, such as ProbLog, reduce the expensive inference to an efficient Weighted Model Counting. To do so ProbLog employs a sequence of transformation steps, called an *inference pipeline*. Each step in the probabilistic inference pipeline is called a *pipeline component*. The choice of the mechanism to implement a component can be crucial to the performance of the system. In this paper we describe in detail different ProbLog pipelines and investigate which are the crucial components with respect to the efficiency. Our main contributions are the thorough analysis of ProbLog inference pipelines and the introduction of new pipelines, one of which performs very well on our benchmarks.

In this document we describe in detail our benchmarks and report our full results. Then we discuss our results and determine the dependency between the implementation of the components and the overall pipeline performance.

Keywords : ProbLog, Implementation, Inference Pipeline, System Architecture, Experiments.

Implementation and Performance of Probabilistic Inference Pipelines: Data and Results

Dimitar Shterionov, Gerda Janssens
Computer Science, KULeuven
Celestijnenlaan 200A, bus 2402, Heverlee, Belgium
firstname.lastname@cs.kuleuven.be

March 6, 2015

Abstract

In order to handle real-world problems, state-of-the-art probabilistic logic and learning frameworks, such as ProbLog, reduce the expensive inference to an efficient Weighted Model Counting. To do so ProbLog employs a sequence of transformation steps, called an *inference pipeline*. Each step in the probabilistic inference pipeline is called a *pipeline component*. The choice of the mechanism to implement a component can be crucial to the performance of the system. In this paper we describe in detail different ProbLog pipelines and investigate which are the crucial components with respect to the efficiency. Our main contributions are the thorough analysis of ProbLog inference pipelines and the introduction of new pipelines, one of which performs very well on our benchmarks.

In this document we describe in detail our benchmarks and report our full results. Then we discuss our results and determine the dependency between the implementation of the components and the overall pipeline performance.

1 Probabilistic Inference with ProbLog

ProbLog [7, 12] is a general purpose Probabilistic Logic Programming (PLP) language. It extends Prolog with probabilistic facts which encode uncertain knowledge. Probabilistic facts have the form $p_i :: f_i$, where p_i is the probability label of the fact f_i. Prolog rules define the logic consequences of the probabilistic facts. Figure 1 shows a probabilistic graph and its encoding as a ProbLog program. The fact `0.6::e(a, b).` expresses that the edge between nodes `a` and `b` exists with probability 0.6.

An atom which unifies with a probabilistic fact is called a *probabilistic atom*. An atom which unifies with the head of a rule is called a *derived atom*. The sets of probabilistic and derived atoms of a ProbLog program should be disjoint. In the example of Figure 1 `e(a, b)` is a probabilistic atom, while `p(a, b)` is derived.

A probabilistic atom can be either true with the probability of the corresponding probabilistic fact or false with (1−the probability). A choice of the truth value of such an atom is called an *atomic choice*. The atomic choices of all probabilistic atoms define a *total choice*. For n probabilistic facts there are 2^n total choices. Each total choice defines a (unique) model of the ProbLog program called a *possible world*. A ProbLog program specifies a probability distribution on possible worlds according to the Distribution Semantics [21].

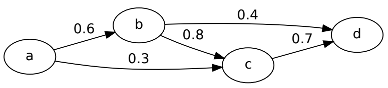
Formally, given a ProbLog program, let $\Omega = \{\omega_1, .., \omega_N\}$ be the set of possible worlds of that program, where $N = 2^n$ and n is the number of probabilistic facts of the ProbLog program. Given that only probabilistic atoms have probabilities we see a single possible world ω_i as the tuple (ω_i^+, ω_i^-), where ω_i^+ is the set of probabilistic atoms in ω_i which are true and ω_i^- the set of probabilistic atoms which are false. Intuitively, the union $\omega_i^+ \cup \omega_i^-$ is the set of all possible ground probabilistic atoms of the ProbLog program with a specific truth value assignment as defined by the corresponding total choice. The intersection $\omega_i^+ \cap \omega_i^-$ is the empty set. Probabilistic atoms are seen as independent random variables. Then a ProbLog program defines a distribution over possible worlds as given in Equation 1 where p_i denotes the probability of the atom a_i.

$$P(\omega_i) = \prod_{a_j \in \omega_i^+} p_j \prod_{a_j \in \omega_i^-} (1 - p_j) \tag{1}$$

A query atom q is true in a subset of the possible worlds: $\Omega^q \subseteq \Omega$. Each $\omega_i^q \in \Omega^q$ has a corresponding probability as computed by Equation 1. The (success or *marginal*) probability of q is the sum of the probabilities of all worlds in which q is true:

$$P(q) = \sum_{i, \omega_i \models q} P(\omega_i) \tag{2}$$

Example 1 *The query* `p(a, d)` *for the program in Figure 1 is true if there is at least one path between nodes* `a` *and* `d`*. This holds in* 15 *out of the* $2^4 = 32$ *possible worlds, shown in the following table together with their probability.*



a) A probabilistic graph

```
0.6::e(a, b).    0.3::e(a, c).    0.8::e(b, c).    0.4::e(b, d).    0.7::e(c, d).
p(X, Y):- e(X, Y).                p(X, Y):- e(X, X1), p(X1, Y).
```

b) A ProbLog program.

Figure 1: A probabilistic graph and its encoding as a ProbLog program. The `p/2` predicate defines the "path" relation between two nodes: a path exists, if two nodes are connected by an edge or via a path to an intermediate node.

Possible World ($\omega_i \models q$)	*e(a,b)*		*e(a,c)*		*e(b, c)*		*e(b, d)*		*e(c, d)*		`p(a, d)` *is true*
ω_1	T	0.6	T	0.3	T	0.8	T	0.4	T	0.7	0.04032
ω_2	T	0.6	T	0.3	T	0.8	T	0.4	F	0.3	0.01728
ω_3	T	0.6	T	0.3	T	0.8	F	0.6	T	0.7	0.06048
ω_5	T	0.6	T	0.3	F	0.2	T	0.4	T	0.7	0.01008
ω_6	T	0.6	T	0.3	F	0.2	T	0.4	F	0.3	0.00432
ω_7	T	0.6	T	0.3	F	0.2	F	0.6	T	0.7	0.01512
ω_9	T	0.6	F	0.7	T	0.8	T	0.4	T	0.7	0.09408
ω_{10}	T	0.6	F	0.7	T	0.8	T	0.4	F	0.3	0.04032
ω_{11}	T	0.6	F	0.7	T	0.8	F	0.6	T	0.7	0.14112
ω_{13}	T	0.6	F	0.7	F	0.2	T	0.4	T	0.7	0.02352
ω_{14}	T	0.6	F	0.7	F	0.2	T	0.4	F	0.3	0.01008
ω_{17}	F	0.4	T	0.3	T	0.8	T	0.4	T	0.7	0.02688
ω_{19}	F	0.4	T	0.3	T	0.8	F	0.6	T	0.7	0.04032
ω_{21}	F	0.4	T	0.3	F	0.2	T	0.4	T	0.7	0.00672
ω_{23}	F	0.4	T	0.3	F	0.2	F	0.6	T	0.7	0.01008
										$\sum =$	**0.54072**

We refer to the task of computing the marginal probability of a query as the *MARG* task. ProbLog can also compute the probability of a query given evidence on some ground atoms, i.e. the conditional probability of the query given that the evidence holds – the *COND* task. Evidence is a set of atoms E for which the truth values e are given: $E = e$.

Example 2 *For the program in Figure 1, the query* `p(a, d).` *and the evidence* `e(a, b)` $= false$ *ProbLog computes the conditional probability* $P($`p(a, d)`$|$`e(a, b)`$= false) = 0.21$.

Performing MARG and COND inference revolve around the same mechanism. There are, though, some differences in how the truth values given to the evidence atoms are used. That is why we perform separate experiments on MARG and COND inference.

ProbLog also can handle multiple queries simultaneously without reconsulting the same program. That is, ProbLog can compute simultaneously the probabilities $P(q|E = e)$ for $q \in Q$, where Q is a set of queries.

1.1 Weighted Model Counting by Knowledge Compilation

Enumerating the possible worlds of a ProbLog program and computing the (marginal) probability of a query according to Equation 2 is a straightforward approach for probabilistic inference. Because the number of possible worlds grows exponentially with the increase of the number of probabilistic facts in a ProbLog program, this approach is considered impractical.

In order to avoid the expensive enumeration of possible worlds the inference mechanism of ProbLog uses knowledge compilation and an efficient weighted model counting method. Model Counting is the process of determining the number of models of a formula φ: $\#SAT(\varphi) = \sum_{m_i \in SAT(\varphi)} 1$. The *Weighted* Model Count (WMC) of a formula φ is the sum of the weights that are associated with each model of φ: $WMC(\varphi) = \sum_{m_i \in SAT(\varphi)} w(m_i)$, where w is a weight function that associates a weight with a model. For a given ProbLog program L with a set of possible worlds Ω the WMC of a formula φ coincides with Equation 2 when there is a bijection between the models (and their weights) of φ and the possible worlds (and their probabilities) in Ω: $\Omega \equiv SAT(\varphi)$ and $p(\omega_i) = w(m_i)$ for all $(\omega_i, m_i) \in (\Omega, SAT(\varphi))$.

The task of Model Counting (and also its specialization Weighted Model Counting) is in general a $\#P$-complete problem. Its importance in SAT and in the Statistical Relational Learning and Probabilistic Logic and Learning communities has lead to the development of efficient algorithms [5] which have found their place in ProbLog. By using knowledge compilation the actual WMC can be computed linearly to the size of the compiled (arithmetic) circuit [5, Chapter12].

1.2 Inference Pipeline

In order to transform a ProbLog inference task into a WMC problem an initial ProbLog program (together with queries and evidence) needs to be compiled into a Boolean formula with special properties that allows to efficiently perform WMC. To do so ProbLog uses a sequence of transformation steps, called an *inference pipeline.* There are four main transformation steps, i.e. *components* that compose an inference pipeline: *Grounding*, *Boolean formula conversion*, *Knowledge Compilation* and *Evaluation*. The grounding generates a propositional instance of the input ProbLog program. It ignores the probabilistic information of that program, i.e. the probability label of each probabilistic fact. Second, this propositional instance is converted to an equivalent with respect to the models Boolean formula. Third, the Boolean formula is compiled into a *negation normal form* (NNF) with certain properties which allow efficient model counting. Finally, this NNF is converted to an arithmetic circuit which is associated with the probabilities of the input program and weighted model counting is performed.

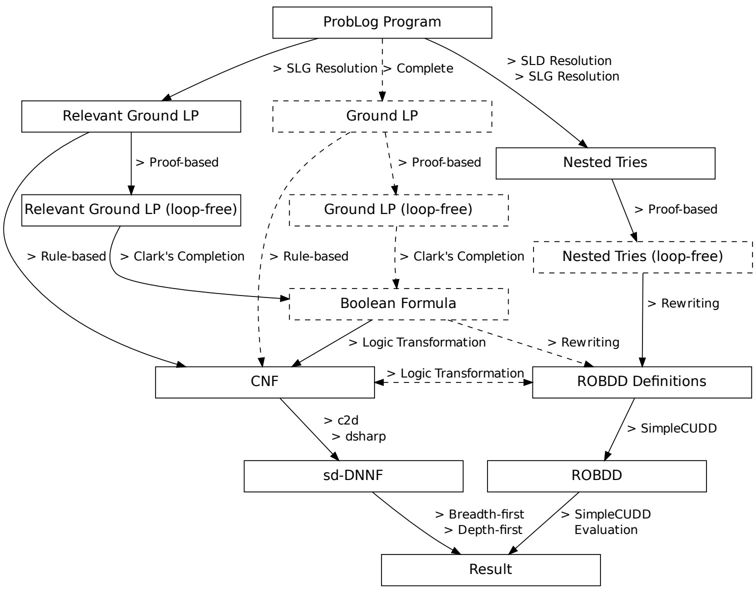


Figure 2: ProbLog pipelines. Directed edges define the processes which take place for each inference step. Solid edges define a ProbLog1 or ProbLog2 pipeline. Dashed edges state a transformation which is not included in neither of the ProbLog1 or ProbLog2 pipelines. Nodes specify input/output formats. Dashed nodes indicate complementary data formats. The input ProbLog program may contain query and evidence atoms.

Each component can use different tools or algorithms to perform the necessary transformation, as long as the input/output requirements between components are respected. For example, ProbLog1 [7] uses knowledge compilation to *Reduced Ordered Binary Decision Diagrams* (ROBDDs) [1] in order to reduce the inference task to a tractable problem. Later, [9] illustrates an approach for ProbLog inference by compilation to a *smooth, deterministic, Decomposable Negation Normal Form* (sd-DNNF) [6]. Figure 2 gives an overview of the different approaches that can be used to implement a component and how they can be linked in order to form an inference pipeline.

1.2.1 Grounding

A naive grounding approach is to generate all possible instances of the initial ProbLog program according to all the values the variables can be bound to. Such complete grounding may result in extremely big ground programs and therefore is not considered in any working pipeline. It is more efficient with respect to the size of the grounding and the time for its generation to focus on the part of the ProbLog program which is relevant to an atom of interest. A ground ProbLog program is relevant to an atom q if it contains only relevant atoms and rules. An atom is relevant if it appears in some *proof* of q. A ground rule is relevant with respect to q if its head is a relevant atom and its body consists of relevant atoms. It is safe to confine to the ground program relevant to q because the models of the relevant ground program are the same as the models of the initial ProbLog program which entail the atom q. That is, the relevant ground program captures the distribution $P(q)$ entirely (proof of correctness can be found in [8]).

To determine the relevant grounding a natural mechanism is SLD resolution. Each successful SLD derivation for a query q determines one proof of q – a conjunction of ground literals. Naturally, all proofs to a query form a disjunction and therefore, can be represented as a Boolean formula in DNF. An SLD derivation may be infinite, e.g., in case of cyclic programs. In order to avoid complications caused by cycles SLG resolution [2] (that is, SLD with tabling) can be used instead. Cycles can be detected by introducing additional code to the input ProbLog program in order to store and compare intermediate results. Such a mechanism though, can become slow and is susceptible to user errors. That is why tabling (i.e. SLG resolution) is preferable for ProbLog inference.

We distinguish between two representations of the relevant grounding of a ProbLog program. ProbLog1 uses the **nested trie structure** as an intermediate representation of the collected proofs. If SLD resolution is used (that is, no tabling is invoked)[1] there is only one trie which corresponds to the SLD tree. ProbLog2 considers the **relevant ground logic program** with respect to a set of (query) atoms.

1.2.2 Boolean Circuit of the Grounding

Logic Programs (LP) use the Closed World Assumption (CWA), which basically states that if an atom cannot be proven to be true, it is false. In contrast, First-Order logic (FOL) has different semantics: it does not rely on the CWA. Consider the (FOL) theory $\{q \leftarrow p\}$ which has three models: $\{\neg q, \neg p\}$, $\{q, \neg p\}$ and $\{q, p\}$. Its syntacticly equivalent LP (`q :- p.`) has only one model, namely $\{\neg q, \neg p\}$. In order to generate a Boolean Circuit from nested tries, as it is the case in ProbLog1, or a relevant ground LP (ProbLog2) it is required to make the transition from LP semantics to FOL semantics. When the grounding does not contain cycles it suffices to take the Clark's completion of that program

[1]ProbLog1 allows the user to select whether to use tabling or not. ProbLog2 always uses tabling.

[10, 11]. When the grounding contains cycles it is proven that the Clark's completion does not result in an equivalent Boolean circuit [11]. To handle cyclic groundings ProbLog employs one of two methods. The first one (referred to as the **proof-based** approach) [15] basically removes proofs containing cycles as they do not contribute to the probability. Furthermore, this approach is query-directed, i.e. it considers a set of queries and traverses their proofs. The second one (referred to as the **rule-based** approach) is inherited from the field of Answer Set Programming. It rewrites a rule with cycles to an equivalent rule and introduces additional variables in order to disallow cycles [11].

Once the cycles are handled, ProbLog1 rewrites the formula encoded in the Nested Tries as **ROBDD definitions**. A ROBDD definition [15] is a formula with a head and a body, linked with equivalence. The body of a ROBDD definition contains literals and/or heads of other ROBDD definitions combined by conjunctions or disjunctions. The logic operators are translated to arithmetic functions. A ROBDD script is a set of ROBDD definitions.

In the case of ProbLog2, once the cycles are handled the relevant ground LP is converted to a formula in **CNF**. The ground LP can also be rewritten to **ROBDD definitions**.

Example 3 *Consider the Boolean formula:*
$(a \iff (b \wedge c)) \wedge (b \iff (p \vee q)) \wedge (c \iff \neg r)$.
Following are its equivalent representations as a CNF and ROBDD definitions:

CNF:	*ROBDD definitions:*
$(a \vee \neg b \vee \neg c) \wedge (\neg a \vee b) \wedge (\neg a \vee c) \wedge$	`b = p + q`
$(\neg b \vee p \vee q) \wedge (b \vee \neg p) \wedge (b \vee \neg q) \wedge$	`c = ~r`
$(\neg c \vee \neg r) \wedge (c \vee r)$	`a = b * c`

Example 4 *A CNF formula can be translated into ROBDD definitions and vice-versa. The following ROBDD definitions are generated from the CNF in Example 3 and are equivalent to the Boolean formula in Example 3:*

```
ROBDD definitions 2:
L1 = ~b + p + q   L2 = b + ~p        L3 = b + ~q   L4 = c + r
L5 = ~c + ~r      L6 = a + ~b + ~c   L7 = ~a + c   L8 = ~a + b
L9 = L1 * L2 * L3 * L4 * L5 * L6 * L7 * L8
```

The Clark's completion of a cycle-free logic program is a formula similar to the one in Example 3. This formula can easily be converted to CNF as well as in ROBDD definitions. Example 3 shows that a CNF representation of such a formula is less succinct ([6]) than the representation as ROBDD definitions. If though a CNF formula is converted to ROBDD definitions as in Example 4 the ROBDD script blows up in size. For the overall performance of a pipeline it is crucial to avoid components which perform such a transformation without any formula reduction. This phenomenon is discussed among others in [19]. In [9, 8] the authors consider a ProbLog pipeline in which a CNF formula is transformed into ROBDD definitions as shown in Example 4, i.e. a relevant ground LP is first converted to a Boolean circuit in CNF which subsequently is converted to a ROBDD script. Their experiments confirm that such an approach is inefficient for ProbLog inference. In this paper we do not consider inference pipelines which include a transformation from CNF to ROBDD definitions. To the contrary, we introduce a *new pipeline* which transforms the relevant ground program directly into ROBDD definitions avoiding the blow up of the ROBDD script (see Table 1, pipeline $P4$).

1.2.3 Knowledge Compilation and Evaluation

Knowledge compilation is the process in which a Boolean Circuit is compiled to a negation normal form (NNF) with certain properties [6]. For correct inference ProbLog requires the compiled circuit to be *deterministic*, *decomposable* and *smooth*. In ProbLog's inference pipelines two target compilation languages have been exploited so far: (i) **ROBDDs** [1] common for ProbLog1 (and MetaProbLog [14, Chapter 6]) and (ii) **sd-DNNFs** [6] employed by ProbLog2.

To compile a Boolean circuit as a ROBDD ProbLog implementations use **SimpleCUDD** (`www.cs.kuleuven.be/~theo/tools/simplecudd.html`). Compiling to sd-DNNF is done with the **c2d** [3, 4] or **dsharp** [17] compilers.

After the Knowledge compilation step, the compiled circuit is traversed in order to compute the probabilities (i.e. the WMC) for the given query(ies) – the evaluation step. ProbLog employs two approaches to traverse sd-DNNFs: **Breadth-First** and **Depth-First**[2]) and one to traverse ROBDDs.

Sections 1.2.1 to 1.2.3 describe the components of the two mainstream ProbLog pipelines – ProbLog1 and ProbLog2. The subprocesses which are used in these pipelines constitute a set of interchangeable components which may form other working pipelines. Figure 2 gives an overview of the possible ProbLog pipelines. The link between different components depends on the compatibility of the output of a preceding subprocess with the input requirements of the next one. For example, c2d cannot compile ROBDD definitions but requires CNFs. Earlier it was shown that some pipelines are certain to perform worse than others: pipelines with (naive) complete grounding; pipelines in which a CNF is converted to ROBDD Definitions (cf. Section 1.2.2). In addition, we prefer using SLG resolution for grounding instead of SLD resolution in order to infinite proofs caused by cycles. This leaves the 14 pipelines shown in Table 1. $P4$ and $P9..P12$ are previously unexploited pipelines for ProbLog inference.

2 Benchmarks

1. The **Alzheimer** benchmark set [7] is build from a real-world biological dataset of Alzheimer genes. The data is represented as a directed probabilistic graph with 11530 edges and 5220 nodes. We used 17 subgraphs with increasing

[2] To invoke one of these two options in ProbLog2 one specifies either the *fileoptimized* (default) for the Breadth-First implementation or *python* for the Depth-First implementation as evaluation options.

Index	Grounding Form	Conversion	Compilation	Evaluation
$P0$	Ground LP	Proof-Based	c2d	Breadth-First
$P1$	Ground LP	Proof-Based	c2d	Depth-First
P2	Ground LP	Proof-Based	dsharp	Breadth-First
$P3$	Ground LP	Proof-Based	dsharp	Depth-First
P4	Ground LP	Proof-Based	SimpleCUDD	SimpleCUDD
$P5$	Ground LP	Rule-Based	c2d	Breadth-First
$P6$	Ground LP	Rule-Based	c2d	Depth-First
$P7$	Ground LP	Rule-Based	dsharp	Breadth-First
$P8$	Ground LP	Rule-Based	dsharp	Depth-First
$P9$	Nested Tries	Proof-Based	c2d	Breadth-First
$P10$	Nested Tries	Proof-Based	c2d	Depth-First
$P11$	Nested Tries	Proof-Based	dsharp	Breadth-First
$P12$	Nested Tries	Proof-Based	dsharp	Depth-First
P13	Nested Tries	Proof-Based	SimpleCUDD	SimpleCUDD

Table 1: Pipelines used in the experiments. Pipelines $P2$ and $P13$ (in bold) are the default pipelines of ProbLog2 and MetaProbLog respectively. The $P4$ pipeline is a newly proposed pipeline which is one of the best-performing, according to our experiments.

```
0.3::red(0, SG, SB); 0.3::green(0, SG, SB); 0.4::blue(0, SG, SB) <- true.
0.3::red(1, SG, SB); 0.3::green(1, SG, SB); 0.4::blue(1, SG, SB) <- SG < 2, SB < 3, red(0, 0, 0).
0.3::red(1, SG, SB); 0.3::green(1, SG, SB); 0.4::blue(1, SG, SB) <- SG < 2, SB < 3, SGNew is SG + 1, green(0, SGNew, SB).
0.3::red(1, SG, SB); 0.3::green(1, SG, SB); 0.4::blue(1, SG, SB) <- SG < 2, SB < 3, SBNew is SB + 1, blue(0, SG, SBNew).

win(P):-red(P, 0, 0).
win(P):-green(P, 0, 0).
win(P):-blue(P, 0, 0).
```

Figure 3: A ProbLog program encoding the Balls benchmarks, using annotated disjunctions.

sizes and without duplicate edges extracted from the initial graph. We used 6 different path queries for each of the (sub)graphs. With each combination *query-graph* we associate one ProbLog program.
2. The **Balls** benchmark set, presented in [23], contains ProbLog programs encoding a game in which a player draws colorful balls (red, green and blue) from different bags one bag after another. The player can choose only one ball per bag. To be able to select from a bag the previous selections should fulfill certain conditions. The different options for a bag are encoded as annotated disjunctions [24, 16]. We use 40 different queries that compute the probability a ball is selected from a specific bag. Each query is associated with a separate ProbLog program.
3. The probabilistic **Dictionary** benchmark set ([22]) includes around 250 different words from the English language. They are linked to each according to a similarity measure expressed with a probability (probability 1.0 states that two words mean exactly the same; probability 0.0 that two words do not mean the same). The probabilities are computed according to two approaches: (i) the algorithm presented in [22] and (ii) MSR (`http://cwl-projects.cogsci.rpi.edu/msr/`). They form an *incomplete* probabilistic graph. For 30 of the words their meaning is also given. We use 65 randomly selected queries which look for the probability that two words have the same meaning even if an explicit link has not been defined. There are more than 7000 possible combinations. Each query is associated with one ProbLog program.
4. [8] introduces the probabilistic **Grid** as a special case of a probabilistic graph. We use a grid with 25×25 nodes. Each node $n_{x,y}$ is connected by a directed edge to the nodes $n_{x+1,y}$, $n_{x,y+1}$ and $n_{x+1,y+1}$. We use different queries $path(1, n_{x,x})$ where $x = 3, .., 25$.
5. The **Les Miserables** [13] originally is a deterministic dataset presenting the relations of the characters from the same-name novel who appeared in the same chapter. The data was shifted to a probabilistic setting by calculating the probability that two randomly selected characters will appear in the same chapter i.e. a *tie* relation. We use 68 benchmark programs each containing one query. A single query asks for the probability of a tie between two characters.
6. **Smokers** [18] is a dataset which expresses a friend network. Each person can smoke either because of stress or because he/she is *influenced* by a friend who smokes. In a ProbLog program from the "Smokers" benchmark set, the `influence` relations are encoded as probabilistic facts. We use programs with an increasing number of people: from 3 until 100 which is indicative for the size of the program. One benchmark program contains multiple queries and evidence. A single query asks the probability that a person smokes. We use this set for both computing the marginal and the conditional probabilities.
7. The **WebKB** benchmark set is built upon a dataset from a collective classification domain in which university webpages are classified according to their textual content (`http://www.cs.cmu.edu/~webkb/`) [9]. All the probabilities are learned from data [10]. We use 98 programs containing different sets of queries and evidence atoms. We do both MARG and COND inference. We compute is the marginal (or the conditional) probability a classification is correct (given the evidence holds).

The programs in the "Alzheimer", "Dictionary", "Grid", "Les Miserables", "Smokers" and "WebKB" benchmark are similar to the one used to encode the probabilistic graph in Figure 1. The program in Figure 3 shows a different probabilistic network used in the "Balls" benchmark set. It employs annotated disjunctions [24] to specify exclusive choices. Annotated disjunctions provide an alternative and more intuitive (that probabilistic facts) encoding of uncertain events with multiple outcomes. ProbLog translates internally the annotated disjunctions into probabilistic graphs.

All benchmarks used in this work can be found at `http://people.cs.kuleuven.be/~dimitar.shterionov/benchmarks_` pipelines.zip

3 Experimental Results

We tested the 14 pipelines (see Table 1) on the 7 benchmark sets discussed in the previous section. We executed the MARG task on all of the 7 benchmark sets[3] and the COND task on the last 2 sets – "Smokers" and "WebKB".

In our experiments, we measure the run times of each subprocess (grounding, conversion, compilation and evaluation) while performing the MARG or the COND task for the given query(ies) and evidence. Because the c2d compiler is non-deterministic (cf. [3]), i.e. for the same CNF the compiled sd-DNNFs may differ, we run all tests 5 times and report the average run time. Previous tests with the c2d compiler within ProbLog pipelines have shown that the average time for 5 runs gives a realistic estimate on its performance. We set a time-out of 540 seconds for each run.

We expect that our results show how the different pipelines perform compared to each other. Then we can assess the impact that a certain component has on the overall pipeline performance. We could then identify which component(s) is crucial for a ProbLog pipeline and what the reasons are.

In Section 3.1 we present the run time for each pipeline on the benchmarks. Section 3.2 summarizes our results in tables in order to determine which are the best performing pipelines. A thorough discussion then follows in Section 4.

3.1 Time Diagrams

We present the total runtime (the sum of the grounding, conversion, compilation and evaluation times) of each pipeline for a benchmark program executing MARG or COND inference; the lower the time is, the better. The reason to focus only on the total run time is that any change in the performance of two pipelines which share all but one component will be due to the different components. To get an idea of the impact of individual components we compare the result for pipelines which differ by one component. For example, comparing pipelines $P0-P8$ to pipelines $P9-P13$ will determine the effect of the the two different grounding approaches.

In a diagram each horizontal line is associated with one program and shows the runtime of each pipeline (x-axis) executing the MARG or the COND task on that program; the colors of each line are automatically generated in accordance to the complexity of the program. The complexity is measured by the size of the dependency graph representing the ground ProbLog program. The black line parallel to the x-axis indicates the 540^{th} second, that is, the time-out. We use a logarithmic scale for the time axis (the y-axis).

3.1.1 MARG Inference

[3]For the last two benchmark sets "Smokers" and "WebKB" in order to compute the MARG task and not the COND we ignore any evidence given in a program.

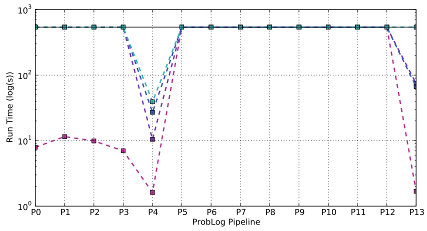


Figure 4: Run times for the Alzheimer set, query $q1$.

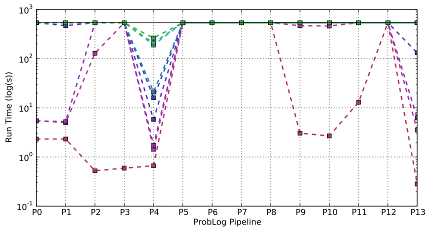


Figure 5: Run times for the Alzheimer set, query $q2$.

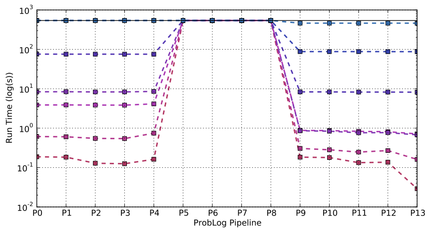


Figure 6: Run times for the Alzheimer set, query $q3$.

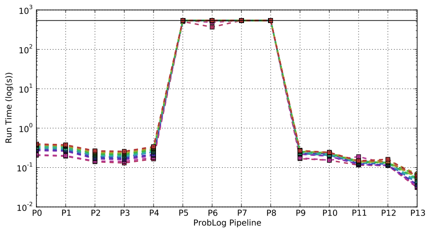


Figure 7: Run times for the Alzheimer set, query $q4$.

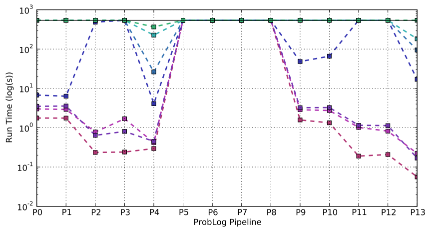


Figure 8: Run times for the Alzheimer set, query $q5$.

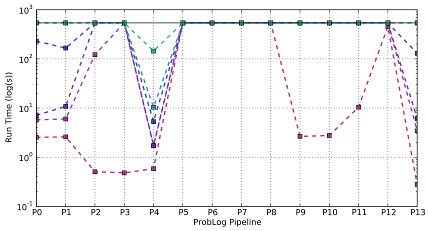


Figure 9: Run times for the Alzheimer set, query $q6$.

3.1.2 COND Inference

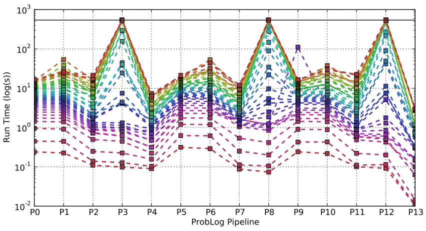


Figure 10: Run times for the Balls set.

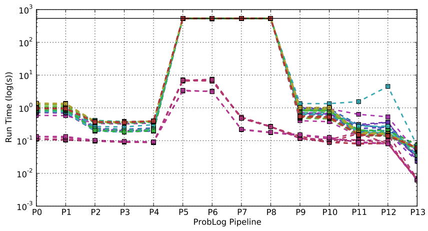


Figure 11: Run times for the Dictionary set.

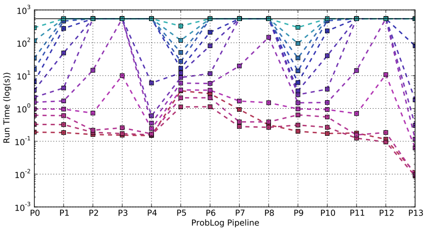


Figure 12: Run times for the Grid set.

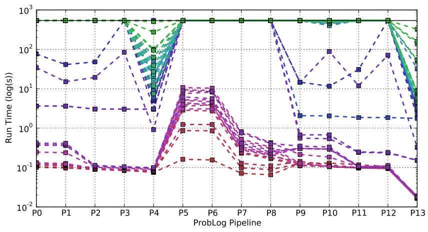


Figure 13: Run times for the Les Miserables set.

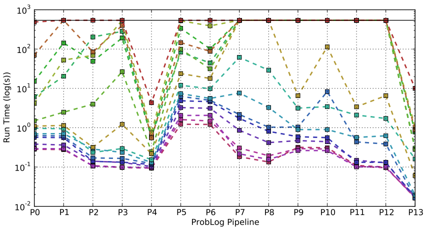


Figure 14: Run times for the Smokers set.

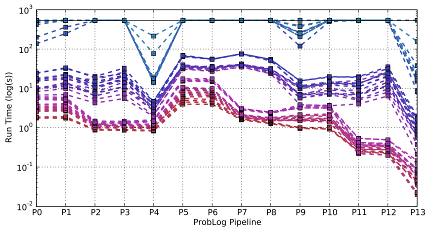


Figure 15: Run times for the WebKB set.

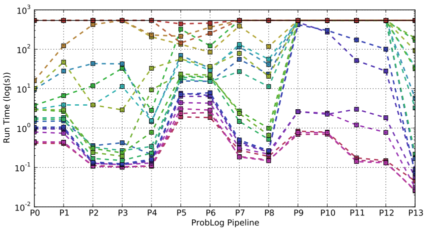


Figure 16: Run times for the Smokers set.

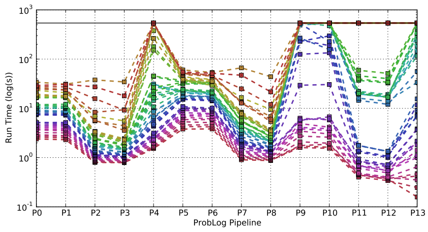


Figure 17: Run times for the WebKB set.

3.2 Best-performing Pipelines

Tables 2, 3, 4, 5, 6, 7 and 9 show an ascending ordering of the pipelines running MARG inference on the corresponding examples according to the (total) runtime for the “Alzheimer”, “Balls”, “Dictionary”, “Grid”, “Les Miserables”, “Smokers” and “WebKB” benchmark sets. That is, 1^{st} indicates the pipeline that performs best (in lowest time); 2^{nd} indicates the second best pipeline, and so forth. Tables 8 and 10 show an ascending ordering of the pipelines running COND inference for the example programs in the “Smokers” and the “WebKB” benchmarks. For readability the index label $'P'$ is omitted, that is, each pipeline is associated only with a number. The empty cells indicate that no pipeline has successfully executed the inference task within the time out limit (540 seconds).

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
alzheimer_q1	graph05	4	13	3	0	2	1								
alzheimer_q1	graph06	4	13												
alzheimer_q1	graph07	4	13												
alzheimer_q1	graph08	4													
alzheimer_q2	graph05	13	2	3	4	1	0	10	9	11					
alzheimer_q2	graph06	4	13	1	0	2	10	9							
alzheimer_q2	graph07	4	1	0	13										
alzheimer_q2	graph08	4	13	1											
alzheimer_q2	graph09	4													
alzheimer_q2	graph10	4													
alzheimer_q2	graph11	4													
alzheimer_q2	graph12	4													
alzheimer_q2	graph13	4													
alzheimer_q3	graph01	13	3	2	11	12	4	10	1	9	0				
alzheimer_q3	graph05	13	11	12	10	9	3	2	1	0	4				
alzheimer_q3	graph06	13	12	11	10	9	3	2	0	1	4				
alzheimer_q3	graph07	13	11	12	10	9	2	3	0	1	4				
alzheimer_q3	graph08	13	12	11	10	9	4	3	1	2	0				
alzheimer_q3	graph09	13	12	10	11	9									
alzheimer_q3	graph11	11	13	9	12	10									
alzheimer_q4	graph05	13	12	11	3	2	10	4	9	1	0	6	5		
alzheimer_q4	graph06	13	12	3	2	10	9	4	11	1	0	6			
alzheimer_q4	graph07	13	12	11	3	2	10	4	9	1	0				
alzheimer_q4	graph08	13	12	11	3	2	10	4	9	1	0				
alzheimer_q4	graph09	13	12	11	3	2	4	10	9	1	0				
alzheimer_q4	graph10	13	12	11	3	2	10	9	4	1	0				
alzheimer_q4	graph11	13	12	11	3	2	10	9	4	1	0				
alzheimer_q4	graph12	13	11	12	3	2	10	9	4	1	0				
alzheimer_q4	graph13	13	12	11	3	10	2	9	4	1	0				
alzheimer_q4	graph14	13	12	11	3	2	10	9	4	1	0				
alzheimer_q4	graph15	13	12	11	3	2	10	9	4	1	0				
alzheimer_q4	graph16	13	12	11	3	10	2	9	4	1	0				
alzheimer_q4	graph17	13	12	11	10	3	2	9	4	1	0				
alzheimer_q4	graph18	13	11	12	10	3	2	9	4	1	0				
alzheimer_q4	graph19	13	12	11	10	3	2	9	4	1	0				
alzheimer_q4	graph20	13	11	12	10	3	2	9	4	1	0				
alzheimer_q5	graph05	13	11	12	2	3	4	10	9	1	0				
alzheimer_q5	graph06	13	4	2	12	11	3	10	9	1	0				
alzheimer_q5	graph07	13	4	2	3	12	11	9	10	0	1				
alzheimer_q5	graph08	4	1	0	13	9	10	2							
alzheimer_q5	graph09	4	13												
alzheimer_q5	graph10	13	4												
alzheimer_q5	graph12	4													
alzheimer_q6	graph05	13	3	2	4	0	1	9	10	11	12				
alzheimer_q6	graph06	4	13	0	1	2									
alzheimer_q6	graph07	4	13	0	1										
alzheimer_q6	graph08	4	13	1	0										
alzheimer_q6	graph09	4													
alzheimer_q6	graph10	4													

Table 2: Ascending order of pipelines according to their total runtime for each program of the “Alzheimer” benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
balls	test1	13	8	7	4	12	3	11	2	10	1	0	9	6	5
balls	test2	13	4	8	11	7	12	3	2	9	10	1	0	6	5
balls	test3	13	4	12	8	11	3	2	7	10	9	1	0	6	5
balls	test4	13	4	8	12	3	11	2	7	1	10	9	0	6	5
balls	test5	13	4	12	11	3	2	8	7	10	9	1	0	6	5
balls	test6	13	4	12	11	3	2	8	7	1	0	10	9	6	5
balls	test7	13	4	12	11	3	8	2	7	10	9	1	0	6	5
balls	test8	13	4	12	11	3	2	8	7	10	9	1	0	6	5
balls	test9	13	4	11	12	3	2	8	7	10	9	1	0	6	5
balls	test10	13	4	11	3	2	8	12	7	10	1	0	6	5	9
balls	test11	13	4	11	3	2	7	12	8	10	9	1	0	5	6
balls	test12	13	4	11	3	2	7	8	10	9	1	0	12	6	5
balls	test13	13	4	11	2	7	3	8	10	9	1	0	6	5	12
balls	test14	13	4	11	2	7	3	9	10	0	1	8	6	5	12
balls	test15	13	4	11	2	7	9	10	1	0	5	6	3	8	12
balls	test16	13	4	11	2	7	3	9	10	0	1	6	5	8	12
balls	test17	13	4	2	7	11	9	10	0	1	5	6	3	8	12
balls	test18	13	4	2	7	11	9	0	10	1	5	6	3	8	12
balls	test19	13	4	11	7	2	9	0	10	1	5	6	3	8	12
balls	test20	13	4	11	7	2	9	0	1	10	5	6	3	12	8
balls	test21	13	4	7	11	2	9	0	10	1	5	6	3	8	12
balls	test22	13	4	7	11	2	9	0	10	1	5	6	3	12	
balls	test23	13	4	7	2	11	9	0	1	10	5	6	3		
balls	test24	13	4	7	11	2	9	0	1	5	10	6	3		
balls	test25	13	4	7	11	2	9	0	5	10	1	6			
balls	test26	13	4	7	11	2	9	0	10	5	1	6			
balls	test27	13	4	7	2	11	9	0	5	6	1	10			
balls	test28	13	4	7	11	2	9	0	5	10	1	6			
balls	test29	13	4	7	11	9	2	0	5	1	10	6			
balls	test30	13	4	7	2	11	9	0	5	1	10	6			
balls	test31	13	4	7	2	11	9	0	5	10	6	1			
balls	test32	13	4	7	2	11	9	0	5	6	1	10			
balls	test33	13	4	7	11	2	9	0	5	10	6	1			
balls	test34	13	4	7	11	9	0	2	5	10	1	6			
balls	test35	13	4	7	11	9	0	2	5	1	6	10			
balls	test36	13	4	7	2	9	11	0	5	1	10	6			
balls	test37	13	4	7	11	9	0	2	5	10	6	1			
balls	test38	13	4	7	9	0	11	2	5	1	10	6			
balls	test39	13	4	7	9	0	2	5	11	1	10	6			
balls	test40	13	4	7	9	0	5	2	11	1	10	6			

Table 3: Ascending order of pipelines according to their total runtime for each program of the "Balls" benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
dictionary	q1	13	12	11	4	10	3	2	1	0	9	8	7	5	6
dictionary	q2	13	12	4	3	10	2	1	0	9	11	8	7	5	6
dictionary	q3	13	4	3	2	12	11	1	10	0	9	8	7	5	6
dictionary	q4	13	4	3	2	1	10	0	9	12	11	8	7	5	6
dictionary	q5	13	11	12	4	3	10	2	1	0	9	8	7	6	5
dictionary	q6	13	12	4	10	3	2	1	11	0	9	8	7	6	5
dictionary	q7	13	12	4	11	3	2	10	1	0	9	8	7	6	5
dictionary	q8	13	12	11	4	3	2	10	1	0	9	8	7	6	5
dictionary	q9	13	3	2	4	11	12	1	0	10	9				
dictionary	q10	13	3	4	12	2	11	10	9	1	0				
dictionary	q11	13	12	11	3	2	4	10	9	1	0				
dictionary	q12	13	12	11	3	2	4	10	9	1	0				
dictionary	q13	13	12	11	3	2	4	10	9	0	1				
dictionary	q14	13	12	11	3	2	4	10	9	1	0				
dictionary	q15	13	11	12	3	2	4	10	9	1	0	6			
dictionary	q16	13	12	11	3	2	4	10	9	1	0				
dictionary	q17	13	12	11	3	2	4	10	9	1	0				
dictionary	q18	13	12	11	3	2	4	10	9	1	0				
dictionary	q19	13	12	11	3	2	4	10	9	0	1				
dictionary	q20	13	3	2	4	1	0	10	9	11	12				
dictionary	q21	13	12	11	3	2	4	10	9	0	1				
dictionary	q22	13	3	11	4	12	2	10	9	1	0				
dictionary	q23	13	12	11	3	2	4	10	9	1	0				
dictionary	q24	13	12	11	3	2	4	10	9	1	0				
dictionary	q25	13	11	12	3	2	4	9	10	1	0				
dictionary	q26	13	11	12	4	2	3	10	1	9	0				
dictionary	q27	13	4	3	2	11	12	1	0	10	9				
dictionary	q28	13	12	11	3	2	4	10	9	1	0				
dictionary	q29	13	12	11	3	2	4	9	10	0	1				
dictionary	q30	13	11	12	3	2	4	10	9	0	1				
dictionary	q31	13	12	11	3	2	4	9	10	1	0				
dictionary	q32	13	12	11	4	3	2	10	9	1	0				
dictionary	q33	13	11	4	3	2	12	10	9	1	0				
dictionary	q34	13	12	11	3	2	4	10	9	1	0				
dictionary	q35	13	12	11	3	4	2	10	9	1	0				
dictionary	q36	13	12	11	3	2	4	10	9	1	0				
dictionary	q37	13	12	11	3	2	4	10	9	0	1				
dictionary	q38	13	3	12	2	4	11	1	0	10	9				
dictionary	q39	13	12	11	3	2	4	10	9	1	0				
dictionary	q40	13	12	11	3	2	4	10	9	1	0				
dictionary	q41	13	11	12	3	2	4	10	9	1	0				
dictionary	q42	13	12	11	3	2	4	10	9	1	0				
dictionary	q43	13	12	11	3	2	4	9	10	1	0				
dictionary	q44	13	3	2	4	11	12	1	0	10	9				
dictionary	q45	13	12	11	3	2	4	10	9	1	0				
dictionary	q46	13	12	11	3	2	4	9	10	0	1				
dictionary	q47	13	12	11	3	2	4	10	9	1	0				
dictionary	q48	13	12	11	3	2	4	10	9	1	0				
dictionary	q49	13	12	11	2	3	4	10	9	1	0				
dictionary	q50	13	4	3	2	12	11	1	0	10	9				
dictionary	q51	13	12	11	3	2	4	9	10	1	0				
dictionary	q52	13	12	11	3	2	4	9	10	1	0				
dictionary	q53	13	3	4	2	11	12	10	9	1	0				
dictionary	q54	13	12	11	3	4	2	0	1	10	9				
dictionary	q55	13	3	2	4	12	11	1	0	10	9				
dictionary	q56	13	12	11	3	2	4	10	9	1	0				
dictionary	q57	13	12	11	3	2	4	10	9	1	0				
dictionary	q58	13	12	11	3	4	2	10	9	0	1				
dictionary	q59	13	11	12	3	4	2	10	9	1	0				
dictionary	q60	13	12	11	3	2	4	10	9	0	1				
dictionary	q61	13	3	2	4	11	12	1	0	10	9				
dictionary	q62	13	12	11	3	2	4	10	9	1	0				
dictionary	q63	13	12	11	3	4	2	10	9	0	1				
dictionary	q64	13	12	11	4	3	2	9	10	0	1				

Table 4: Ascending order of pipelines according to their total runtime for each program of the “Dictionary” benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
grid	test_1	13	12	4	3	2	10	11	1	0	9	8	7	6	5
grid	test_2	13	12	11	4	3	2	8	10	7	9	1	0	5	6
grid	test_3	13	11	4	12	2	3	8	7	10	1	0	9	5	6
grid	test_4	13	4	11	2	10	1	0	9	8	7	6	5	3	12
grid	test_5	13	4	9	10	0	1	6	5	11	2	7	8		
grid	test_6	13	4	0	9	10	1	5	6						
grid	test_7	13	9	0	4	5	10	1	6						
grid	test_8	9	0	5	13	6	10	1							
grid	test_9	9	0	5	1	10									
grid	test_10	0	9	5											
grid	test_11	9	0	5											
grid	test_12	0	9	5											

Table 5: Ascending order of pipelines according to their total run time for each program of the “Grid” benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
les_miserables	test_1	13	8	7	4	3	2	12	1	11	0	10	9	6	5
les_miserables	test_2	13	4	3	8	2	7	1	0	12	11	9	10	6	5
les_miserables	test_3	13	4	3	2	12	11	8	10	1	7	0	9	6	5
les_miserables	test_4	13	4	3	2	12	11	10	1	9	0	8	7	6	5
les_miserables	test_5	13	4	3	12	2	11	10	9	1	0	8	7	5	6
les_miserables	test_6	13	4	3	2	12	11	10	1	0	9	8	7	6	5
les_miserables	test_7	13	4	3	12	2	11	10	9	1	0	8	7	5	6
les_miserables	test_8	13	4	3	12	2	11	10	9	1	0	8	7	5	6
les_miserables	test_9	13	4	3	2	12	11	10	1	9	0	8	7	6	5
les_miserables	test_11	13	4	3	12	2	11	10	9	1	0	8	7	6	5
les_miserables	test_12	13	4	12	3	11	2	8	10	7	9	1	0	6	5
les_miserables	test_13	13	4	3	11	2	12	8	10	9	7	0	1	5	6
les_miserables	test_14	13	4	12	3	11	2	8	10	9	1	7	0	6	5
les_miserables	test_15	13	4	12	3	2	11	8	10	9	1	0	7	6	5
les_miserables	test_23	13	4	12	3	11	2	10	9	8	1	0	7	6	5
les_miserables	test_24	13	12	11	10	9	3	4	2	1	0				
les_miserables	test_25	13	12	11	9	10	4	3	2	0	1				
les_miserables	test_26	13	4	11	9	1	2	0	12	3	10				
les_miserables	test_27	13	4												
les_miserables	test_28	13	4	10											
les_miserables	test_29	13	4	10	9										
les_miserables	test_30	13	10	9	4	11	1	2	0						
les_miserables	test_31	13	4	9											
les_miserables	test_32	13	12	11	10	9									
les_miserables	test_33	13	4	10											
les_miserables	test_34	13	4												
les_miserables	test_35	13	4	10											
les_miserables	test_36	13	4												
les_miserables	test_37	13	4	10											
les_miserables	test_38	4	13	10											
les_miserables	test_39	13	4	10											
les_miserables	test_40	13	4	10											
les_miserables	test_41	13	4	10											
les_miserables	test_42	13	4												
les_miserables	test_43	13	4	10											
les_miserables	test_44	13	4	10											
les_miserables	test_45	13	4	10											
les_miserables	test_48	13	4	10											
les_miserables	test_49	13	4	10											
les_miserables	test_50	13	4												
les_miserables	test_51	13	4	9											
les_miserables	test_52	4	13												
les_miserables	test_53	13	4												
les_miserables	test_54	13	4												
les_miserables	test_55	13	4												

Table 6: Ascending order of pipelines according to their total runtime for each program of the "Les Miserables" benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
smokers	smokers-3-6	13	4	3	12	2	11	8	7	1	0	10	9	6	5
smokers	smokers-4-8	13	4	12	3	11	2	8	10	1	0	9	7	6	5
smokers	smokers-5-10	13	4	12	3	2	11	1	0	8	10	9	7	6	5
smokers	smokers-6-12	13	4	3	11	12	2	8	7	10	9	1	0	5	6
smokers	smokers-7-14	13	4	3	2	12	11	1	0	8	9	7	6	5	10
smokers	smokers-8-16	13	4	11	12	3	2	1	10	0	9	8	7	6	5
smokers	smokers-9-18	13	4	3	2	11	12	0	1	9	10	8	6	5	7
smokers	smokers-10-20	4	13	2	3	1	0	12	11	9	10	6	5	8	7
smokers	smokers-11-22	4	13	0	2	6	1	3	5						
smokers	smokers-12-24	13	4	0	1	6	5	2	3						
smokers	smokers-13-26	4	13	0	1	2	6	5							
smokers	smokers-14-28	4	13	0	1	2	3	6	5						
smokers	smokers-15-30	4	13	0	2	6	5	3							
smokers	smokers-16-32	13	4	2	0	1	3	11	9	12	6	5	10		
smokers	smokers-17-34	4	13	0	6										

Table 7: Ascending order of pipelines according to their total runtime for each program of the "Smokers" benchmark set executing the MARG task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
smokers	smokers-3-6	13	3	4	2	12	11	8	7	1	0	10	9	6	5
smokers	smokers-4-8	13	3	2	4	12	11	8	7	1	0	9	10	5	6
smokers	smokers-5-10	13	3	2	4	8	7	1	12	0	11	10	9	6	5
smokers	smokers-6-12	13	3	2	4	12	11	8	7	1	0	10	9	6	5
smokers	smokers-7-14	13	3	2	4	8	7	0	1	6	5	12	11	10	9
smokers	smokers-8-16	13	3	2	4	8	7	1	0	12	10	9	11	6	5
smokers	smokers-9-18	13	3	2	4	8	7	0	1	5	6	12	11	10	9
smokers	smokers-10-20	13	4	2	3	0	1	6	5	8	7				
smokers	smokers-11-22	4	0	2	1	3	6	13	8	5	7				
smokers	smokers-12-24	4	13	0	1	6	8	3	2	5	7				
smokers	smokers-13-26	4	0	1	2	3	6	13	5						
smokers	smokers-14-28	3	2	4	0	1	13	8	6	5	7				
smokers	smokers-15-30	3	2	4	8	1	7	0	6	5	13				
smokers	smokers-16-32	3	2	13	4	8	7	1	0	5	6				
smokers	smokers-17-34	3	2	8	7	1	0	4	6	5	13				
smokers	smokers-18-36	3	2	0	8	4	6	1	5	7	13				
smokers	smokers-19-38	6	8	5	4	7									
smokers	smokers-20-40	5	6												
smokers	smokers-21-42	5	6												
smokers	smokers-22-44	0	1	5	4	6	2								

Table 8: Ascending order of pipelines according to their total runtime for each program of the “Smokers” benchmark set executing the COND task.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
webkb	webkb-3	13	12	11	4	3	2	10	9	8	7	1	0	6	5
webkb	webkb-4	13	12	11	4	3	2	10	9	8	7	1	0	6	5
webkb	webkb-5	13	12	11	4	3	2	9	10	8	7	0	1	6	5
webkb	webkb-6	13	12	11	3	4	2	8	10	9	7	0	1	6	5
webkb	webkb-7	13	12	11	4	3	2	10	9	8	7	1	0	5	6
webkb	webkb-8	13	12	11	4	3	2	8	9	10	7	0	1	6	5
webkb	webkb-9	13	12	11	4	3	2	10	9	8	7	1	0	5	6
webkb	webkb-10	13	12	11	3	2	4	10	9	8	7	0	1	5	6
webkb	webkb-11	13	12	11	4	3	2	8	10	7	9	0	1	5	6
webkb	webkb-12	13	12	11	3	4	2	8	10	9	7	0	1	5	6
webkb	webkb-13	13	12	11	3	4	2	8	9	10	7	1	0	6	5
webkb	webkb-14	13	12	11	3	2	4	8	7	9	10	0	1	6	5
webkb	webkb-15	13	12	11	3	2	4	8	9	10	7	1	0	6	5
webkb	webkb-16	13	12	11	3	2	4	8	10	9	7	1	0	6	5
webkb	webkb-17	13	12	11	3	2	4	8	7	10	9	0	1	6	5
webkb	webkb-18	13	12	11	3	2	4	8	7	10	9	1	0	6	5
webkb	webkb-19	13	11	12	3	2	4	8	7	9	10	1	0	6	5
webkb	webkb-20	13	12	11	3	2	4	8	7	10	9	1	0	5	6
webkb	webkb-21	13	12	11	3	2	4	8	7	10	9	1	0	5	6
webkb	webkb-22	13	4	11	2	9	10	3	12	0	1	8	6	5	7
webkb	webkb-23	13	4	10	9	11	2	3	12	0	1	8	6	5	7
webkb	webkb-24	13	4	9	10	11	2	0	1	3	12	8	6	5	7
webkb	webkb-25	13	4	9	11	10	2	0	1	3	12	8	6	5	7
webkb	webkb-26	13	4	9	11	2	0	10	3	12	1	8	6	5	7
webkb	webkb-27	13	4	9	10	11	0	2	1	3	12	8	6	5	7
webkb	webkb-28	13	4	11	2	9	10	0	3	12	1	8	6	5	7
webkb	webkb-29	13	4	9	11	10	2	0	1	12	3	8	6	5	7
webkb	webkb-30	13	4	9	2	11	10	0	3	1	12	8	5	6	7
webkb	webkb-31	13	4	11	9	10	2	0	3	1	12	8	6	5	7
webkb	webkb-32	13	4	9	11	10	2	0	3	1	12	8	6	5	7
webkb	webkb-33	13	4	9	10	11	2	0	1	3	12	8	6	5	7
webkb	webkb-34	13	4	2	9	11	10	0	3	12	1	8	6	5	7
webkb	webkb-35	13	4	9	2	11	10	3	0	12	1	8	6	5	7
webkb	webkb-36	13	4	9	11	10	2	0	1	3	12	8	6	5	7
webkb	webkb-37	13	4	9	0										
webkb	webkb-38	13	4	9	0	1	10								
webkb	webkb-39	13	4	0	9	1									
webkb	webkb-40	13	4	9	0										
webkb	webkb-41	4	13	9	0										
webkb	webkb-42	4	13	9											
webkb	webkb-43	4	13	9											
webkb	webkb-44	4	13	9											
webkb	webkb-46	4													

Table 9: Ascending order of pipelines according to their total runtime for each program of the “WebKB” benchmark set executing the MARG task.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	2	0	0	0	33	0	0	0	0	3	0	1	0	229
2^{th}	3	2	1	11	115	0	0	0	2	3	1	16	82	22
3^{th}	13	3	14	21	8	5	0	22	2	22	16	93	26	0
4^{th}	8	6	32	77	26	0	1	4	2	11	15	26	12	3
5^{th}	5	8	85	28	9	1	4	5	0	15	15	27	13	0
6^{th}	8	7	52	14	50	3	1	3	2	18	21	21	9	0
7^{th}	32	16	16	9	5	3	2	0	29	25	63	3	4	0
8^{th}	16	25	4	7	11	16	2	17	1	62	33	5	3	0
9^{th}	28	80	1	6	0	2	2	3	10	18	33	5	8	0
10^{th}	67	35	1	1	3	8	5	15	0	32	15	3	10	0
11^{th}	17	24	0	0	0	1	25	3	34	4	6	0	0	0
12^{th}	23	9	0	9	0	5	19	21	1	6	2	0	1	0
13^{th}	0	0	0	1	0	38	43	0	7	0	0	0	2	0
14^{th}	0	0	0	0	0	42	19	17	1	1	1	0	9	0
Total:	222	215	206	184	260	124	123	110	91	220	221	200	179	254

Table 11: The number of benchmark programs for which a pipeline performs best, second best and so forth without timing out and executing the MARG task. The total number of programs shows how many programs have been executed with the corresponding pipeline and does not include programs for which the pipeline timeouts.

Set	Benchmark	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}	7^{th}	8^{th}	9^{th}	10^{th}	11^{th}	12^{th}	13^{th}	14^{th}
webkb	webkb-3	13	12	11	3	2	8	7	4	10	9	1	0	5	6
webkb	webkb-4	13	12	11	3	2	8	7	10	4	9	1	0	6	5
webkb	webkb-5	12	13	11	3	2	8	7	4	9	10	0	1	6	5
webkb	webkb-6	12	11	13	3	2	8	7	4	10	9	1	0	6	5
webkb	webkb-7	13	12	11	3	2	8	7	4	10	9	1	0	6	5
webkb	webkb-8	12	11	13	3	2	8	7	4	10	9	1	0	6	5
webkb	webkb-9	12	11	3	2	8	7	13	4	1	0	10	9	6	5
webkb	webkb-10	12	11	3	2	13	8	7	4	10	9	1	0	5	6
webkb	webkb-11	12	11	3	2	13	8	7	4	10	9	0	1	6	5
webkb	webkb-12	12	11	3	2	8	7	13	4	1	0	10	9	6	5
webkb	webkb-13	12	11	3	2	8	7	13	4	0	1	9	10	6	5
webkb	webkb-14	12	11	3	2	8	7	4	13	1	0	9	10	6	5
webkb	webkb-15	12	11	3	2	8	7	13	4	1	0	6	5	9	10
webkb	webkb-16	12	11	3	2	8	7	4	13	0	1	6	5	9	10
webkb	webkb-17	12	3	2	11	8	7	4	1	0	13	6	5	10	9
webkb	webkb-18	12	3	2	11	8	7	4	13	1	0	6	5	9	10
webkb	webkb-19	3	2	12	8	11	7	4	13	1	0	6	5	10	9
webkb	webkb-20	3	2	12	8	11	7	4	1	0	13	6	5	10	9
webkb	webkb-21	3	12	2	8	11	7	4	1	0	13	6	5		
webkb	webkb-22	3	2	8	7	1	0	4	12	11	6	5	13		
webkb	webkb-23	3	2	8	7	4	1	0	12	11	6	5	13		
webkb	webkb-24	3	2	8	7	4	0	1	12	6	5	11	13	9	
webkb	webkb-25	3	2	8	7	1	0	12	11	6	5	4	13		
webkb	webkb-26	3	2	8	7	0	1	12	6	11	5	4	13		
webkb	webkb-27	3	2	8	7	1	0	12	4	11	6	5	13		
webkb	webkb-28	3	2	8	7	1	0	12	6	11	5	4	13		
webkb	webkb-29	3	2	8	7	1	0	4	12	11	6	5	13	10	
webkb	webkb-30	3	2	8	7	1	4	0	12	11	6	5	13		
webkb	webkb-31	3	2	8	7	1	0	12	4	11	6	5	13		
webkb	webkb-32	3	2	8	7	0	1	4	12	11	6	5	13		
webkb	webkb-33	3	2	8	7	1	0	6	12	5	11	4			
webkb	webkb-34	3	2	8	7	1	0	4	5	6	12	11	13		
webkb	webkb-35	3	2	8	7	1	0	4	6	5	12	11			
webkb	webkb-36	3	2	8	7	1	0	6	5	4	12	11			
webkb	webkb-37	3	2	8	7	0	1	6	5	4					
webkb	webkb-38	3	2	8	7	1	0	6	5	4					
webkb	webkb-39	3	2	8	7	1	0	6	5						
webkb	webkb-40	3	2	8	7	0	1	6	5	4					
webkb	webkb-41	3	2	8	7	1	0	6	5						
webkb	webkb-42	3	2	8	7	1	0	6	5	4					
webkb	webkb-43	3	2	8	7	1	0	6	5						
webkb	webkb-44	3	2	8	7	1	0	6	5	4					
webkb	webkb-45	1	0	3	2	8	6	5	7						
webkb	webkb-46	3	2	8	7	1	0	6	5						
webkb	webkb-47	3	8	2	7	1	0	6	5						
webkb	webkb-48	8	2	3	7	1	0	6	5						
webkb	webkb-49	3	8	2	7	1	0	6	5						
webkb	webkb-50	3	8	2	0	1	7	6	5						

Table 10: Ascending order of pipelines according to their total runtime for each program of the "WebKB" benchmark set executing the COND task.

Table 11 and Table 13 summarize the order results from the previous tables and show for the total number of benchmark programs for which each pipeline performed best, second best and so forth for the MARG task and for the COND task respectively. Table 12 and Table 14 show the number of benchmark programs for which each pipeline performs best, second best and so forth relative to the total number of programs in a benchmark set for the MARG task and the COND task respectively. For example, pipeline $P8$ performs second best for two benchmarks – one from the "Balls" and one from the "Les Miserables" sets. There are 45 and 40 benchmarks which have been successfully executed by at least one pipeline in the "Balls" and "Les Miserables" sets respectively. Then we compute the relative number of programs for which $P8$ performs second best as $1/45 + 1/40 = 0.05$.

The total and the relative (to the total number of programs in a benchmark set) number of timeouts for a pipeline executing the MARG and the COND tasks are shown in Table 15 and Table 16 respectively.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	0.17	0	0	0	3.71	0	0	0	0	0.25	0	0.14	0	7.73
2^{th}	0.25	0.25	0.11	0.45	3.6	0	0	0	0.05	0.25	0.02	0.85	2.56	2.3
3^{th}	1.18	0.39	0.88	0.98	0.26	0.42	0	0.55	0.05	0.69	0.48	2.74	1.11	0
4^{th}	0.69	0.62	1.1	2.21	0.86	0	0.07	0.1	0.05	0.32	1.03	0.91	0.66	0.34
5^{th}	0.32	0.34	2.81	0.99	0.16	0.08	0.28	0.12	0	1.07	0.75	0.81	0.6	0
6^{th}	0.31	0.76	1.67	0.82	1.29	0.16	0.07	0.07	0.05	0.49	1.17	0.7	0.21	0
7^{th}	0.86	0.55	0.77	0.54	0.27	0.17	0.15	0	0.93	1.31	1.63	0.17	0.14	0
8^{th}	0.64	1.0	0.09	0.21	0.69	0.54	0.17	0.56	0.03	1.72	1.12	0.19	0.07	0
9^{th}	0.87	2.8	0.14	0.14	0	0.05	0.05	0.13	0.38	0.61	0.84	0.4	0.22	0
10^{th}	2.47	0.93	0.08	0.02	0.43	0.2	0.17	0.41	0	0.9	0.49	0.05	0.37	0
11^{th}	0.46	0.68	0	0	0	0.07	0.79	0.17	0.87	0.18	0.19	0	0	0
12^{th}	0.65	0.21	0	0.22	0	0.26	0.53	0.61	0.08	0.24	0.09	0	0.03	0
13^{th}	0	0	0	0.08	0	1.11	1.21	0	0.22	0	0	0	0.05	0
14^{th}	0	0	0	0	0	1.19	0.57	0.48	0.03	0.03	0.07	0	0.28	0
Total (relative):	8.87	8.53	7.65	6.66	11.27	4.25	4.06	3.2	2.74	8.06	7.88	6.96	6.3	10.37

Table 12: The relative number of benchmark programs to the total number of benchmarks in a set for which a pipeline performs best, second best and so forth without timing out and executing the MARG task. The total number of programs shows how many programs have been executed with the corresponding pipeline relative to the total number of programs in the benchmark set, and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	1	1	0	35	3	2	1	0	1	0	0	0	13	11
2^{th}	3	1	32	9	1	0	2	0	4	0	0	10	4	2
3^{th}	2	1	14	10	3	2	0	0	25	0	0	4	2	3
4^{th}	2	2	11	7	9	0	0	28	5	0	0	2	0	0
5^{th}	5	25	6	2	3	0	2	1	14	0	0	3	3	2
6^{th}	22	6	1	0	1	0	4	18	9	0	0	3	0	1
7^{th}	5	5	0	1	13	1	16	8	4	0	0	0	5	6
8^{th}	2	5	1	0	13	18	6	4	1	0	1	1	8	4
9^{th}	6	9	0	0	7	9	4	1	1	1	6	9	1	0
10^{th}	9	2	0	0	0	5	9	4	0	7	2	2	3	6
11^{th}	2	6	0	0	4	7	7	0	0	4	5	4	2	0
12^{th}	6	2	0	0	0	7	0	0	0	5	3	3	0	12
13^{th}	0	0	0	0	0	3	14	0	0	4	6	0	0	0
14^{th}	0	0	0	0	0	14	3	0	0	5	3	0	0	0
Total:	65	65	65	64	57	68	68	64	64	26	26	41	41	47

Table 13: The number of benchmark programs for which a pipeline performs best, second best and so forth without timing out and executing the COND task. The total number of programs shows how many programs have been executed with the corresponding pipeline and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
1^{th}	0.05	0.02	0	0.88	0.15	0.1	0.05	0	0.02	0	0	0	0.27	0.46
2^{th}	0.12	0.05	0.81	0.39	0.05	0	0.1	0	0.11	0	0	0.21	0.08	0.07
3^{th}	0.1	0.05	0.52	0.21	0.15	0.1	0	0	0.55	0	0	0.08	0.04	0.09
4^{th}	0.07	0.1	0.29	0.17	0.45	0	0	0.61	0.16	0	0	0.04	0	0
5^{th}	0.13	0.61	0.12	0.1	0.09	0	0.1	0.05	0.44	0	0	0.06	0.15	0.04
6^{th}	0.49	0.15	0.05	0	0.02	0	0.17	0.55	0.22	0	0	0.15	0	0.05
7^{th}	0.19	0.22	0	0.05	0.3	0.02	0.36	0.17	0.2	0	0	0	0.1	0.18
8^{th}	0.1	0.16	0.05	0	0.27	0.46	0.21	0.17	0.05	0	0.02	0.02	0.2	0.08
9^{th}	0.15	0.27	0	0	0.15	0.39	0.11	0.05	0.05	0.02	0.12	0.19	0.05	0
10^{th}	0.27	0.04	0	0	0	0.13	0.25	0.2	0	0.15	0.07	0.07	0.06	0.21
11^{th}	0.04	0.12	0	0	0.08	0.15	0.15	0	0	0.14	0.19	0.08	0.1	0
12^{th}	0.12	0.04	0	0	0	0.15	0	0	0	0.19	0.09	0.15	0	0.25
13^{th}	0	0	0	0	0	0.09	0.41	0	0	0.08	0.18	0	0	0
14^{th}	0	0	0	0	0	0.41	0.09	0	0	0.16	0.06	0	0	0
Total: (relative)	1.83	1.83	1.84	1.8	1.71	2	2	1.8	1.8	0.74	0.73	1.05	1.05	1.43

Table 14: The relative number of benchmark programs to the total number of benchmarks in a set for which a pipeline performs best, second best and so forth without timing out and executing the COND task. The total number of programs shows how many programs have been executed with the corresponding pipeline relative to the total number of programs in the benchmark set, and does not include programs for which the pipeline timeouts.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	46	53	62	84	8	144	145	158	177	48	47	68	89	14
Total (relative):	3.14	3.48	4.35	5.34	0.72	7.76	7.94	8.8	9.28	3.95	4.12	5.04	5.71	1.64

Table 15: Total and relative (to the total number of benchmarks in a set) number of benchmark programs for which MARG inference times out.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
Total:	3	3	3	4	11	0	0	4	4	42	42	27	27	21
Total (relative):	0.15	0.15	0.15	0.2	0.29	0.0	0.0	0.2	0.2	1.25	1.25	0.94	0.94	0.55

Table 16: Total and relative (to the total number of benchmarks in a set) number of benchmark programs for which COND inference times out.

4 Discussion

To determine the influence of the different components on the overall performance we compare the run times of pipelines which differ by only one component. For example, pipeline $P0$ differs from pipeline $P9$ by the grounding component – $P0$ uses the ProbLog2 grounder to determine a ground logic program (Ground LP, in short), while $P9$ the MetaProbLog grounder to Nested Tries.

We discuss the results from our experiments with the MARG task separately from the COND task. This is because computing the conditional probabilities in MetaProbLog (whose components we use to build other pipelines) differs from how conditional probabilities are computed in ProbLog2. The difference is in whether the truth values given to the evidence atoms are exploited.

4.1 MARG Inference

Grounding Comparing pipelines $P0, .., P4$ to $P9, .., P13$ shows that grounding to a Ground LP and grounding to Nested Tries have similar impacts on the performance. Grounding to a Ground LP is done by an SLD resolution-like approach which proves each subgoal while memoizing the ground clauses and atoms participating in a proof. Once a subgoal is proven the relevant clauses and atoms are written into a file – the Ground LP. Collecting the relevant grounding in a Nested Trie structure is done by the SLD (or SLG, when tabling is invoked) resolution of YAP Prolog, adapted to perform an extra bookkeeping of the probabilistic information. Each SLD refutation is stored as a list of ground probabilistic atoms in a trie or a forest of nested tries when tabling is invoked.

There is one drawback of the relevant Ground LP approach which influences the next component, namely the Boolean formula conversion. The relevant grounding of successful subgoals that participate in a body of a clause is included in the ground program even if the body is false.

Example 5 *Consider the program in Fig. 1 and the query* `p(a, b).`. *The minimal set of ground atoms and clauses required to compute the probability of the query is:* `0.6::e(a, b).` *and* `p(a, b):- e(a, b).`. *The Ground LP as computed by the grounding component is:*

```
0.6::e(a,b).
p(a,b) :- e(a,b).
0.8::e(b,c).     0.7::e(c,d).     0.4::e(b,d).     0.3::e(a,c).
```

The Ground LP in Example 5 contains an extra set of ground atoms (the last four ground probabilistic atoms) which do not contribute to the probability computation. The Ground LP associated with each benchmark from the "Grid" set contains unnecessary atoms as in Example 5. To ensure minimal Ground LP one solution is to employ a second traversal in order to remove subgoals which participate in failing derivations. These optimization is unnecessary in the scope of the default ProbLog2 pipeline ($P2$) because the Proof-based Boolean formula conversion implemented for ProbLog2 handles the excessive knowledge appropriately.

Boolean Formula Conversion When comparing pipelines $P0, .., P3$, to $P5, .., P8$ we observe that the Boolean formula conversion has a strong impact on the performance. By itself the time for conversion is not significant but it is the output Boolean formula that strongly influences the next components in the inference pipeline – compilation and evaluation. Knowledge compilation is computationally the most expensive task. The Proof-based approach generates Boolean formulae which are easier to compile (with the c2d, dsharp or SimpleCUDD tools), than the Rule-based approach (compare $P0, .., P3$, to $P5, .., P8$ in Fig. 4..9, Fig. 11 and Fig. 13..15).

Table 11 and Table 12 show that the pipelines that use Rule-based conversion never perform best, that is, never 1^{st} but also not 2^{nd}. The time out results in Table 15 show that pipelines using the Proof-base conversion time out 42% to 59%[4] less than pipelines using the Rule-based approach.

The input for the conversion is the grounding, represented either as a Ground LP or as Nested Tries. As noted earlier, the Ground LP may contain additional ground atoms and clauses. The implementation of the Proof-based conversion which is used with the Ground LP is different from the implementation of the algorithm for Nested Tries. It applies on an AND-OR graph encoding of the Ground LP. First, an AND-OR graph is constructed from the Ground LP. Each ground conjunction is encoded with one AND node that has as children the conjuncts; each ground disjunction is encoded with one OR node with as children the disjuncts; each ground atom of a fact (probabilistic or not) is encoded as a terminal node. Edges express the dependency among nodes, i.e. an edge from a child to a parent AND node states that the atom encoded by the child node is true; an edge from a child to a parent OR node states that the atom or the conjunction encoded by the child node is true and participates in a proof of the (sub)goal expressed by the parent node. A conjunction with at least one false conjunct is false and therefore, neither an AND node is introduced in the AND-OR graph nor associated edges. Second, the Proof-based conversion starts to traverse the graph from a node associated with a query atom and breaks any cycles generating an AND-OR tree. This AND-OR tree does not include the excessive knowledge from the Ground LP. The AND-OR tree is then rewritten to a Boolean formula which is minimal with respect to the Ground LP.

The Rule-based approach though, does build a Boolean formula in CNF from the ground program one clause after another. The extra ground atoms and clauses from the Ground LP are also considered in building the Boolean formula. The number of literals and the number of clauses in the formulae generated by this approach are often significantly larger compared to the Proof-based approach. The extra information in the Ground LP though, does not have a high impact

[4]We use the relative number of timeouts rather than the total number of timeouts in order to determine a more general interval.

on the total performance. This is obvious from the results for the "Grid" benchmarks (Fig. 12) where the Ground LP has extra knowledge similar to the program in Example 5.

For the effectiveness of the conversion of major importance is the presence of cycles in the grounding. We notice (Fig. 10 and Fig. 12) that pipelines using the Rule-based conversion handle the acyclic graphs from the "Balls" and the "Grid" benchmark sets equally well or even better than some of the pipelines using the Proof-based conversion. This is because the conversion does not need to handle any cycles and the Rule-based conversion simply needs to traverse the Ground LP and rewrite it as a Boolean formulae.

These results show that the Boolean formula conversion is crucial for the inference pipeline.

Knowledge Compilation and Evaluation Knowledge Compilation is the computationally most expensive task in a ProbLog inference pipeline. We consider two target compilation languages: sd-DNNFs and ROBDDs. Our experiments show that even though sd-DNNFs are at least as succinct as ROBDDs [6], employing ROBDDs in a ProbLog pipeline results in lower run times and better scalability. Compare $P4$ to $P0$, .., $P3$ and $P13$ to $P9$, .., $P12$.

ROBDDs allow polytime Boolean transformations, i.e. bounded conjunction, bounded disjunction and negation [6]. Therefore, compiling to ROBDDs can be performed in an efficient bottom-up manner. The size of an ROBDD strongly depends on the order in which variables are processed. Dynamic variable reordering [20] allows the transformation of ROBDDs during the compilation stage when new variables are presented. Although variable reordering is NP-complete [1] it ensures an optimal size of the ROBDD. The input ROBDD script should not necessarily be in CNF form in contrast to compilation to sd-DNNFs. A ROBDD script can be substantially smaller than a CNF encoding the same Boolean formula.

In the case of knowledge compilation to sd-DNNFs a pipeline which uses c2d shows better scalability compared to one with dsharp but is slower for the less complex problems. Furthermore, the Breadth-First evaluation approach is in general preferable to the Depth-First approach (compare $P0$ to $P1$ or $P11$ to $P12$), although for the "Balls" benchmarks this evaluation approach performs poorly (see $P3$, $P8$ and $P12$ in Figure 10). The reason is the structure of the graph associated with the Ground LP – low out degree, i.e. 9, long paths from the root to the nodes.

The bottom-up compilation, the dynamic reordering and the succinct representation of the Boolean formula as an ROBDD script are the main factors for ProbLog pipelines with ROBDDs to perform faster than those with sd-DNNFs for the MARG task.

Underlying Implementation We ought to comment also on the implementation of these algorithms. The default MetaProbLog pipeline ($P13$) is implemented in YAP Prolog except for the SimpleCUDD, which is written in C/C++. The ProbLog2 pipelines ($P0$ to $P3$ and $P5$ to $P8$) use a YAP Prolog implementation of the grounder; the Rule-based conversion, dsharp and c2d are implemented in C/C++; the Proof-based conversion, both evaluation approaches and the wrapper that binds all components together are written in Python3. Pipeline $P4$ is based on a ProbLog2 pipeline; it uses SimpleCUDD in a Python3 wrapper. Pipelines $P9$ to $P12$ use the MetaProbLog default implementation and a Python3 script to invoke compilation to sd-DNNF and evaluation.

Using Python as a wrapper is a better solution to constructing ProbLog pipelines as it is more flexible and more modular than Prolog. But for implementing the different components may be slower.

4.2 COND Inference

The conditional probability of a query q given evidence $E = e$ is computed as the ratio $P(q|E = e) = \frac{P(q \wedge E=e)}{P(E=e)}$. First both the nominator and denominator need to be computed separately. Then their division gives the final result. MetaProbLog and ProbLog2 use different approaches when it comes to computing the conditional probabilities. In particular, there are differences regarding the grounding to Nested Tries and compiling to ROBDDs compared to grounding to a Ground LP and knowledge compilation to s-DDNNFs.

Grounding We notice from Figure 16 and Figure 17 that grounding to Nested Tries has a negative effect on the overall performance as compared to grounding to a Ground LP, despite the drawback mentioned earlier. The former grounding method uses the following approach: (i) for a query q and evidence $E = e$ a new query $q^{E=e} = q \wedge E = e$ is created; (ii) $q^{E=e}$ and the atoms in E are proven in order to determine the relevant grounding (stored as nested tries). In the latter case, a query q and the atoms in E are used separately and not in a conjunction to determine the relevant ground program. Although the two grounding approaches generate different groundings – not only the representation but also the ground atoms and clauses may differ, it is the evidence atoms and their predetermined values which have an impact on the next components (and therefore the overall performance). The truth values of the evidence play a significant role for knowledge compilation.

Boolean Formula Conversion The Boolean formula is built by using either the Proof-based or the Rule-based method. In the case of pipelines $P0$ to $P9$ the Boolean formula (either represented as a CNF or as a ROBDD script) is augmented with clauses to state the truth values for the evidence atoms. They often help the knowledge compilation as they may prune parts of the compiled circuit.

Knowledge Compilation and Evaluation From the time diagrams for the MARG and the COND inference we see that in the case of COND inference the pipelines with knowledge compilation to sd-DNNFs perform better than pipelines with knowledge compilation to ROBDDs (compare pipelines $P0$ to $P3$ and $P5$ to $P9$). This contrasts with the results from MARG inference. There are three main reasons: (i) the evidence atoms and their truth values are used during knowledge compilation to sd-DNNFs to optimize the sd-DNNFs; (ii) the number of queries – sd-DNNFs are compiled from one CNF while compilation to ROBDDs generates a forest of ROBDDs for each query (in practice for each $q \wedge E = e$ and $E = e$); (iii) in case the conjunction $q \wedge E = e$ is false ($P(q \wedge E = e) = 0.0$ and therefore $P(q|E = e) = 0.0$) compilation to ROBDDs will still compile the necessary ROBDD to compute the probability, thus it will perform unnecessary operations (this is observed for the "WebKB" benchmark programs where a lot of the queries are false given that the evidence holds). The decreased performance due to compilation to ROBDDs is also confirmed by the results in Table 16.

5 Conclusions and Future Work

In this paper we presented a detailed description of the inference pipelines of ProbLog and analyzed their performance on 7 benchmark sets. Through our analysis we determined that the Boolean formula conversion has a crucial impact on the performance of the inference pipeline for both MARG and COND tasks. We showed that in most of the cases pipelines which use a *Proof-based conversion*, *compilation to sd-DNNF with c2d* and the *Breadth-First evaluation* approach and pipelines which use *Proof-based conversion* and *compilation to ROBDDs* perform better than the rest.

We also showed that for computing the COND task it is crucial how the evidence is handled. Pipelines which use *compilation to sd-DNNF* and *Breadth-First evaluation* outperform the rest.

Our future goals revolve around optimizing the Boolean formula so that the cost for knowledge compilation can be reduced and also improving the method to handle evidence for knowledge compilation with ROBDDs. Furthermore, one of the newly introduced pipelines which combines the grounding of ProbLog2 with the knowledge compilation and evaluation of MetaProbLog via a direct conversion of the (cycle-free) relevant ground LP to ROBDD definitions shows very promising results. To determine its actual place among the different ProbLog implementations we plan to further evaluate its performance on all inference and learning tasks supported by ProbLog.

References

[1] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[2] Weidong Chen, Terrance Swift, and David Scott Warren. Efficient top-down computation of queries under the well-founded semantics. *J. Log. Program.*, 24(3):161–199, 1995.

[3] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In Rina Dechter and Richard S. Sutton, editors, *AAAI/IAAI*, pages 627–634. AAAI Press/MIT Press, 2002.

[4] Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 328–332, 2004.

[5] Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks.* Cambridge University Press, 2009. Chapter 12.

[6] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[7] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: a probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2468–2473. AAAI Press, 2007.

[8] Daan Fierens, Guy Van Den Broek, Joris Renkens, Dimitar Shterionov, Bern Gutmann, Ingo Thon, Gerda Janssens, and Luc de Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming, Special Issue on Probability, Logic and Learning*, 2013.

[9] Daan Fierens, Guy Van den Broeck, Ingo Thon, Bernd Gutmann, and Luc De Raedt. Inference in probabilistic logic programs using weighted CNF's. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*, pages 211–220, 2011.

[10] Bernd Gutmann, Ingo Thon, and Luc De Raedt. Learning the parameters of probabilistic logic programs from interpretations. In *ECML/PKDD (1)*, pages 581–596, 2011.

[11] Tomi Janhunen. Representing normal programs with clauses. In *In Proc. of the 16th European Conference on Artificial Intelligence*, pages 358–362. IOS Press, 2004.

[12] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11:235–262, 2011.

[13] D. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* Addison-Wesley, MA, 1993.

[14] Theofrastos Mantadelis. *Efficient Algorithms for Prolog Based Probabilistic Logic Programming.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, November 2012. Janssens, Gerda (supervisor).

[15] Theofrastos Mantadelis and Gerda Janssens. Dedicated tabling for a probabilistic setting. In Manuel V. Hermenegildo and Torsten Schaub, editors, *ICLP (Technical Communications)*, volume 7 of *LIPIcs*, pages 124–133. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[16] Wannes Meert, Jan Struyf, and Hendrik Blockeel. CP-logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In *Proc. 19th International Conf. of Inductive Logic Programming*, 2009.

[17] Christian Muise, Sheila A. Mcilraith, J. Christopher Beck, and Eric Hsu. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*, 2012.

[18] Luc De Raedt, Paolo Frasconi, Kristian Kersting, and Stephen Muggleton, editors. *Probabilistic Inductive Logic Programming - Theory and Applications*, volume 4911 of *Lecture Notes in Computer Science*. Springer, 2008.

[19] Antoine Rauzy, Eric Châtelet, Yves Dutuit, and Christophe Bérenguer. A practical comparison of methods to assess sum-of-products. *Rel. Eng. & Sys. Safety*, 79(1):33–42, 2003.

[20] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '93, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[21] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming (ICLP'95)*, pages 715–729. MIT Press, 1995.

[22] Dimitar Shterionov and Gerda Janssens. Data acquisition and modeling for learning and reasoning in probabilistic logic environment. In Luis Antunes, H. Sofia Pinto, Rui Prada, and Paulo Trigo, editors, *Proceedings of the 15th Portuguese Conference on Artificial Intelligence*, pages 298–312, 2011.

[23] Dimitar Shterionov, Joris Renkens, Jonas Vlasselaer, Angelika Kimmig, Wannes Meert, and Gerda Janssens. The most probable explanation for probabilistic logic programs with annotated disjunctions. To appear in *Proceedings of the 24th International Conference on Inductive Logic Programming*.

[24] Joost Vennekens, Sofie Verbaeten, Maurice Bruynooghe, and Celestijnenlaan A. Logic programs with annotated disjunctions. In *In Proc. Int'l Conf. on Logic Programming*, pages 431–445. Springer, 2004.